Week 10

- What is an exception ?
- Handling Exceptions
- Recovering from Exceptions
- Exception Hierarchy
- Propagation of Exceptions
- The clause finally
- Re-throwing Exceptions
- Writing our own exceptions

Read
Pages 578-591

What is an Exception?

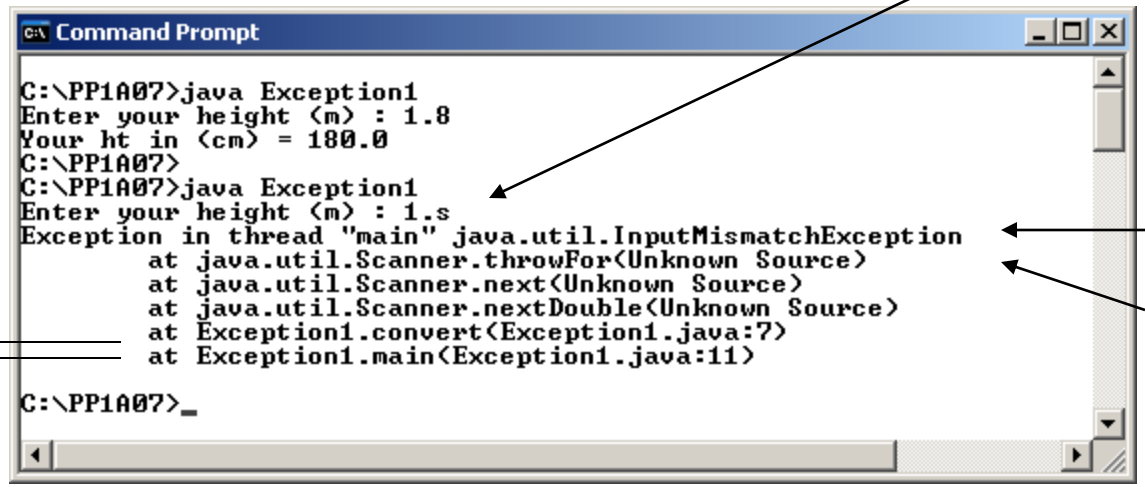
- In the beginning of this course we saw three kinds of errors:
  - Syntax/Compilation Errors
  - Logic Errors
  - Execution/Runtime Errors
- Exception is the mechanism handling Runtime Errors
- An exception occur for number of reasons
  - Accessing outside array bounds → **ArrayIndexOutOfBoundsException**
  - Wrong type of input → **NumberFormatException**
  - Attempting to open a non-existent file → **FileNotFoundException**
  - Problem with keyboard/file input/output → **IOException**
- If exception is not handled program terminates abnormally

Effect of Exception not caught

```
import java.util.*;
public class Exception1
{  public static void convert()
   {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter your height (m) : ");
      double ht = sc.nextDouble();
      System.out.print("Your ht in (cm) = " + ht*100);
   }
   public static void main(String args[])
   {  convert();
   }
}
```

Line 7

Line 11



Cause of exception
Invalid double (1.s)

Exception name

Stack trace

Exceptions

Recall our 2nd week program.

This method is not prepared to handle it!

```
import java.io.*;
public class AddingInts {
   public static void main (String[] args)
                                        throws IOException {
      BufferedReader stdin = new BufferedReader
         (new InputStreamReader (System.in));
      String string1, string2;
      int num1, num2, sum;
      System.out.println ("Input an integer");
      string1 = stdin.readLine();
      num1 = Integer.parseInt (string1);

      System.out.println ("Input another integer");
      string2 = stdin.readLine();
      num2 = Integer.parseInt (string2);
      sum = num1 + num2;
      System.out.print("The sum is: " + sum);
   }
}
```

may throw an IOException

may throw a NumberFormatException

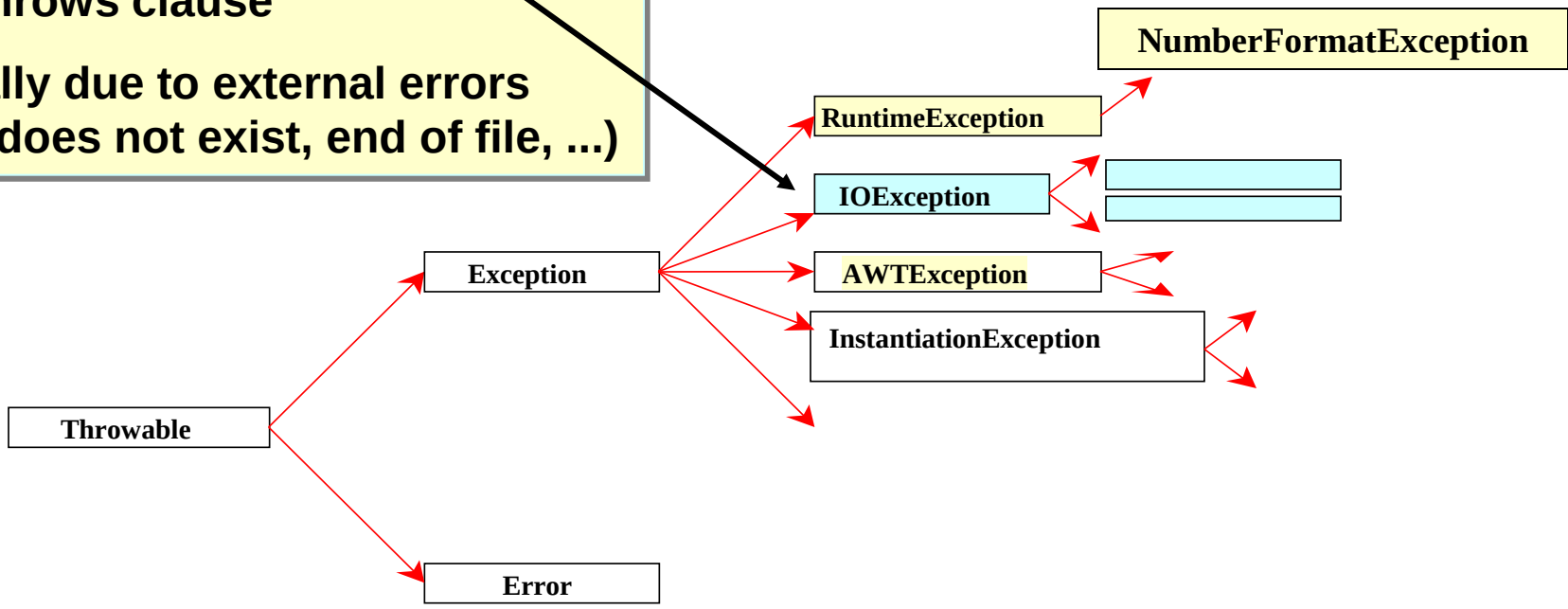
Why is there no throws clause for NumberFormatException ?

Exception classes Hierarchy

Checked Exception

must be caught or listed in the throws clause

Usually due to external errors (File does not exist, end of file, ...)



Last program does not handle the exceptions thrown …

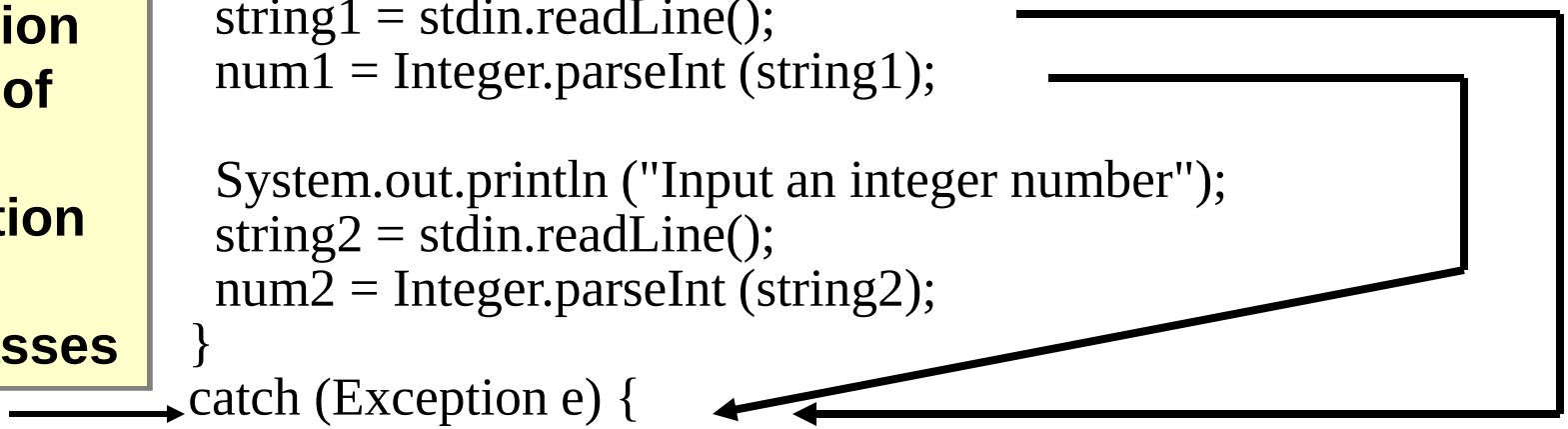
- As the main method does not handle the exception the program aborts …
- Considered Poor programming practice
- Next program catches both type of exceptions but does not attempt to recover from them.

```
import java.io.*;
class AddingInts {
   public static void main (String[] args)  {
      BufferedReader stdin = new BufferedReader
                           (new InputStreamReader (System.in));
      String string1, string2;
      int num1 = 0, num2 = 0, sum = 0;

      try  {
        System.out.println ("Input an integer number");
        string1 = stdin.readLine();
        num1 = Integer.parseInt (string1);

        System.out.println ("Input an integer number");
        string2 = stdin.readLine();
        num2 = Integer.parseInt (string2);
      }
      catch (Exception e) {
         System.err.println("Problem in input.  Exiting ..");
         System.err.println(e);
         System.exit(1);
      }
      sum = num1 + num2;
      System.out.println("The sum is: " + sum);
   }
}
```

Catches any exception that is of type Exception or its subclasses



Refinement 1 … Catches the exceptions separately

```
try  {
   System.out.println ("Input an integer number");
   string1 = stdin.readLine();
   num1 = Integer.parseInt (string1);
   System.out.println ("Input an integer number");
   string2 = stdin.readLine();
   num2 = Integer.parseInt (string2);
 }
 catch (NumberFormatException nfe) { // wrong input
   System.err.println("Invalid input format.  " + "Exiting ...");
   System.err.println(nfe);
   System.exit(1);
 }
 catch (IOException ioe) {
   System.err.println("Problem in input.  Exiting ...");
   System.err.println(ioe);
   System.exit(1);
 }
```

Subclasses of Exception

Still unable to recover from the error ! Next Refinement attempts it.

Refinement 2 … Repeat until user enters valid numbers

Repeat until both numbers are valid

```
boolean valid = false;
do  {
   try  {
      System.out.println ("Input an integer number");
      string1 = stdin.readLine();
      num1 = Integer.parseInt (string1);

      System.out.println ("Input an integer number");
      string2 = stdin.readLine();
      num2 = Integer.parseInt (string2);
      valid = true;
   }
   catch (NumberFormatException nfe) {
      System.err.println("Invalid input: try again");
      System.err.println(nfe);
   }
   catch (IOException ioe) {
      System.err.println("Problem in input.  Exiting ..");
      System.err.println(ioe);
      System.exit(1);
   }
} while (!valid);
```

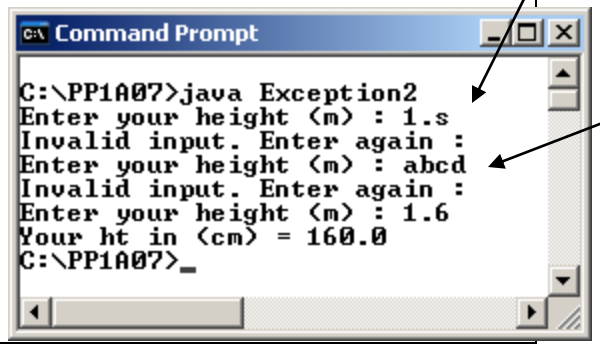
Initially set to false

All okay up to now - (no exceptions thrown) set valid to true

Repeats loop until valid is set to true

Handling the Scanner InputMismatchException

```
import java.util.*;
public class Exception2
{  public static void convert()
   {Scanner sc = new Scanner(System.in);
      boolean done = false;
      do {
         try {
            System.out.print("Enter your height (m) : ");
            double ht = sc.nextDouble();
            System.out.print("Your ht in (cm) = " + ht*100);
            done = true;
         }
         catch(InputMismatchException ex){
            System.out.println("Invalid input. Enter again : ");
            sc.nextLine();
         }
      } while (!done);
   }
   public static void main(String args[])
   {  convert();
   }
}
```



Invalid inputs handled

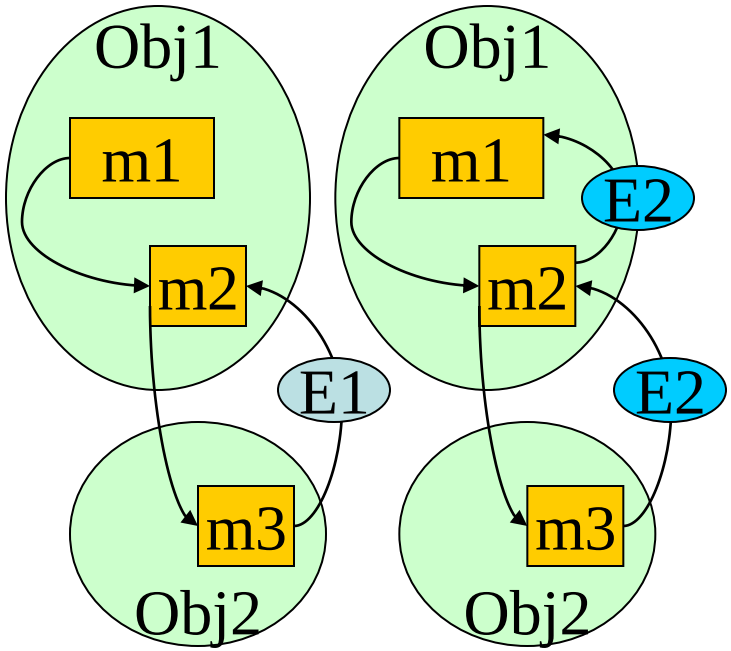
Propagation of Exceptions

Q. What happens If an exception is thrown and there is no catch clause within the method (m3 of object2?)

A. The exception propagates back to the caller of the method

As method m3 propagates exceptions objects of type E1 and E2 it must add the clause throws exception E1, E2

As method m3 propagates exceptions objects of type E1 and E2 it must add the clause throws exception E1, E2



Method m2 catches the exception E1 thrown inside m3

Method m1 catches the exception E2 thrown inside m3

An example

The method below will throw an ArithmeticException if den is passed the value of 0. As it is not caught in the same method it is passed to the calling method and handled there.

```
public static int divide(int num, int den)
                        throws ArithmeticException  {
         return num/den;
}
```

```
import java.io.*;
import java.util.*;
public class CheckException{
    public static void main (String[] args) {
          Scanner console = new Scanner(System.in);
          int i, j, ans=1;
          boolean okay = false;
          System.out.print("Enter numerator : ");
          i = console.nextInt();
         do {
               try {      System.out.print(”Enter demoninator : ");
                    j  = console.nextInt();
                    ans = divide(i,j);
                    okay = true;
                 }
                 catch(ArithmeticException e) {
                      e.printStackTrace();
                      System.out.println("Value for denom cannot be 0");
                 }
         } while ( !okay );
         System.out.println("Answer is " + ans);
   }
   public static int divide(int num, int den) throws ArithmeticException  {
                return num/den;
   }
}
```

→ If den==0 ArithmeticException object will be thrown

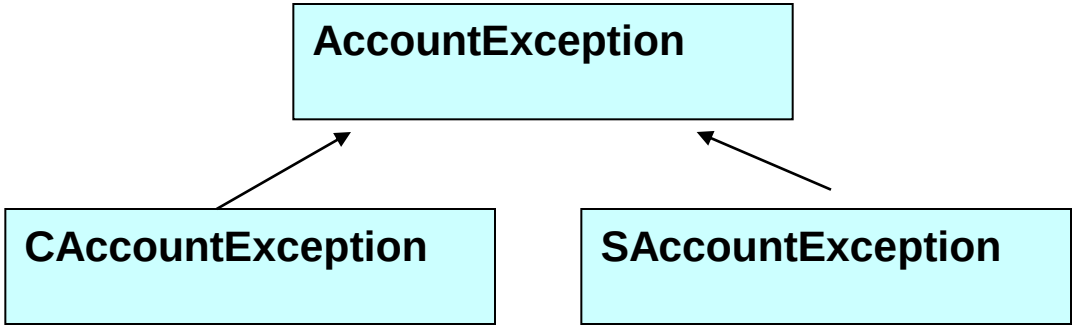
Sample Input/Output

```
Enter numerator          : 8
Enter denominator : 4
Answer is 2
```

```
Enter numerator          :  8
Enter denominator :  0
java.lang.ArithmeticException: / by zero
        at CheckException.main
value for denom cannot be 0
Enter denominator
2
Answer is 4
```

Accounts Example

- As a constructor cannot return a value, it may throw an exception. For example the Account constructor may throw an exception if initial balance passed is negative.
- SAccount may throw an exception if min-amount > initial-balance
- To cater for these we may create our own exception classes. AccountException with subclasses SAccountException and CAccountException.



```
Account acc;
System.out.printn("Specify Account type : ");
String type = console.readLine();
try {
   if ( type.charAt(0) == 'A')  acc = new Account(….)
   else if (type.charAt(0) == 'C')  acc = new CAccount(….)
   else if ( type.charAt(0) == 'S')  acc = new SAccount(….)
}
catch (AccountException ae) { ….}
catch (CAccountException cae) { ….}
catch (SAccountException sae) { ….}
```

All exceptions will be caught by this as it is the superclass - hence rearrange them

```
catch (CAccountException cae) { ….}
catch (SAccountException sae) { ….}
catch (AccountException ae) { ….}
```

Catch the more specialized classes first.

(handled correctly)

The finally construct

You may want to take some action whether or not exception is thrown (such as closing a file).

The finally is used to handle this situation.

In the code below statements B and D may not be reached but statement, C will always be executed.

```
try{  A; // may throw ExType1 or ExType2
      B; //;
}
catch(ExType1 e){
    handling e;
}
finally{
   C; // finalStatements;
}
D ;
```

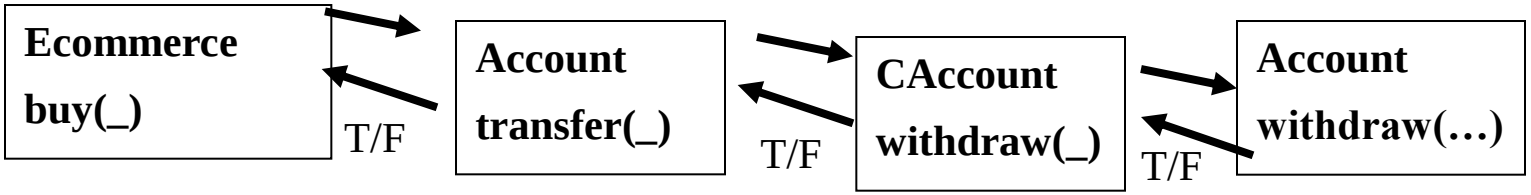
Rethrowing exceptions

When an exception occurs the enclosing method exits immediately, unless the exception is caught.

If we need to perform some tasks before exiting, we can catch it and then *rethrow* it again.

```
try
{   …….
}
catch(TheException e)
{
    perform operations before exits;
    throw e;
}
```

Without exceptions: method must return T/F




```
public void buy(……){  // Ecommerce class
  do {     // may prompt user to enter different values
           …
           if (c1.transfer(c2,amount) == true) {
```

```
public boolean transfer (……){  //  Account class
           if (withdraw(..) == true) {
```

```
public boolean withdraw (……){  //  CAccount class
           if (super.withdraw(..) == true) {
```

```
public boolean withdraw (……){  //  Account class
           if (balance > amt )    {
                      …
                      return true;        }
          else return false;
```

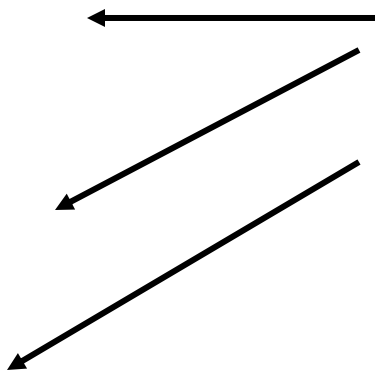
Using the Exception mechanism

```
public void buy(……){  // handles exception
 do {   // may prompt user to enter different values
        try { ...
                  c1.transfer(c2,100)
        catch(WithdrawException we) {  ……..  }

public void transfer (……) throws WithDrawException
{
         withdraw();

public void withdraw (……) throws WithDrawException
{               super.withdraw();

public void withdraw (……) throws WithDrawException
{
        if (balance > amt )  {
                …
        }
        else throw new WithdrawException(…);
```



Propagating the WithDrawException

Writing our Own Exceptions

- The code below defines our own exception class which is thrown when a user attempts to withdraw a –ve *amount* or when the *amount* exceeds the avilable balance.
- This class has *instance variables* for storing information about the error condition.

```
class WithdrawException extends Exception
{
   private String reason;
   private double maxAvailable; //max. withdrawable
   public String getReason() {        return reason; }
   public double maxAvailable() { return maxAvailable;}
   public WithdrawException(String reason) {this.reason = reason;}
   public WithdrawException(String reason,  double maxAvailable) {
      this.reason = reason;
      this.maxAvailable = maxAvailable;
    }
}
```

- Our Account class withdraw can now be redefined to throw this exception whenever any condition is detected.
- Notice the exception object is storing an *error message* and the *maximum amount* that can be withdrawn.
- This information may be used to recover from the error.

```
// if insufficient funds WithdrawException will be thrown
public void withdraw(double amount) throws WithdrawException {
   if (amount < 0 )
      throw new  WithdrawException("Negative Amount not allowed");
   if (balance <  amount)
      throw new WithdrawException("Amount Not Available", balance);
   balance = balance - amount;
}
```

Recovering ...

The program segment next shows how we can recover from this exceptional condition using the WithdrawException object caught.

```
Scanner console = new Scanner(System.in);
double amount;
boolean amountOK = false;
System.out.println("Enter Amount : ");
amount = console.nextDouble();
while (!amountOK)
{  try {
      cDad.withdraw(amount);
      amountOK = true;
   }
   catch (WithdrawException we)
   {  System.out.println(we.getReason());
      if (we.getReason().compareTo("Amount Not Available")==0)
      {  System.out.println("Wish to borrow max amount?(Y/N)");
         if (console.nextLine().charAt(0) == 'Y' )
            amount = we.maxAvailable();
      }
      else
      {  System.out.println("Re-enter Amount : ");
         amount = console.nextDouble();
      }
   }
}
```

Can I avoid checking getCause() ?

Exception : Application

- Recall that in our Part class the supply method returns -1 when there are insufficient stock.
- This is not a good design, as there can be many different reasons why this method failed. For example, the quantity requested may be –ve or too large (above preset limt)
- In this case, throwing an exception is more suitable when one or more error states are detected. Unlike returning a value (such as -1) it can explicitly indicate the cause of error.
- Now modify the supply method to throw exception in two different cases
  - Insufficient stock
  - Incorrect quantity requested (assume valid quantity is in range 1..500
- Modify the driver class to catch and handle the exceptions.

```
import java.util.*;
class Part
{ private String ID;
   private String name;
   private int stLevel;
   private int roLevel;
   private double uPrice;
   public Part(String ID, String name, int sL,
                                        int rL, double uP)
   { this.ID = ID;
      this.name = name;
      stLevel = sL;
      roLevel = rL;
      uPrice = uP;
   }
   public String getID() { return ID; }
   public int getSLevel() { return stLevel; }
   public void replenish(int qty)
   {   stLevel += qty;
   }
   public double supply(int qty)
   {
      if ( stLevel < qty) return -1.0;
      stLevel -= qty;
      if (stLevel < roLevel)
         System.out.println("Place order for " + ID);
      return qty*uPrice;
   }
}
```

```
class TestPart
{  public static void main(String args[])
   {
      Part p = new Part("s123","Axle",120,80,250.0);
      Scanner sc = new Scanner(System.in);
      boolean done = false;
      do {
         System.out.print("Enter qty : ");
         int qty = sc.nextInt();
         double amt = p.supply(qty);
         if ( amt < 0 ) {
            System.out.println("Insufficient Qty");
            System.out.println("Curr. stock = " +
                                        p.getSLevel());
         }
         else {
            System.out.println("Cost for supplying "
                           + p.getID() + " = " + amt);
            done = true;
         }
      } while (!done);
   }
}
```

Changes needed for supply method

- When the quantity requested <= 0 or > 500 throw an Exeption object as in:

```
throw new Exception("Invalid amount");
```

- When the sock level is less than quantity requested throw as in:

```
throw new Exception("Insufficient Stock");
```

- Change the method header to:

```
public double supply(int qty) throws Exception
```

Changes needed in the calling method

- Place the try-catch within a loop which repeats until done is true. Place the call to supply in a try block as in:

```
try {
   double amt = p.supply(qty);
   System.out.println("Cost for supplying " +
           p.getID() + " = " + amt);
   done = true;
}
```

- Catch the Exception and handle it as in:
  - Display the exception
  - Extract the exception message saving it in a string
  - Depending on this message perform appropriate actions

```
catch (Exception e) {
    System.out.println(e);
    String cause = e.getMessage();
    if ( cause.indexOf("Insuff") >=0 ) {
       …          …
    }
    else if ( cause.indexOf("Invalid") >= 0 )
       …
    }
 }
```

Exception : Application

Exception : Application